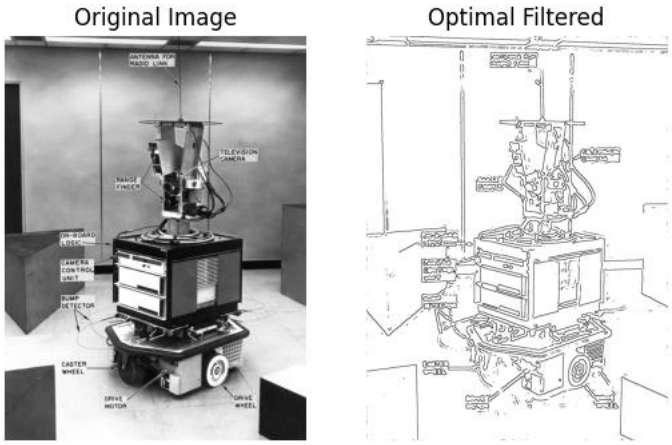
Task 1. The aim is to implement and apply a Laplacian of Gaussian filter to the provided shakey image, in order to discuss its efficacy in edge detection. To start off with, I wrote a function that takes a range of experimental sigma values from 0.5 to 8.0, in steps of 0.1, and for each, applies the Laplacian of Gaussian filter. The image and sigma value is passed to scipy.ndimage package's gaussian_filter method, in essence acting as a smoothing filter that returns a blurred image with reduced noise. I then used NumPy to define a Laplace convolution kernel, which is an approximation of the second derivative of the image. The next step was to convolve the kernel over the smoothed image to retrieve the filtered image for the. Applying the scipy.filters.threshold_otsu() function to the filtered image used Otsu's method to determine the optimal threshold value, which was appended to a list, and the image to another list. After iterating through the test sigma values, the optimal sigma value is taken as the maximum value in the list, which is used to return the corresponding filtered image. The last step was to apply zero crossing, using a function taken from CV&I Lab 2's utils.py package; this detects points in the image where the intensity gradient changes sign, and returns a binary image with value 1 representing these points; in effect showing which pixels are the 'edge pixels' in the image. I then used Matplotlib to display the results.

The optimal sigma value was determined to be 4.9, with the final zero-crossed result being the filtered image shown below. The sigma value controls how much neighbouring pixels contribute to blurring; higher standard deviation means wider Gaussian distribution, meaning the kernel is larger, and the image, put simply, is more blurred. In this case, convolving the Laplacian filter over differently blurred images resulted in differing amounts of noise being falsely detected as edges. Note that the filtered image displays minimal noise/false edges.



Task 2. The aim of this task is to test five different edge detectors on a set of three fluorescing cell images. I wrote 5 separate scripts for this. Each script imports the necessary libraries, reads in the images, and displays the resulting images with Matplotlib (much like with task 1). With the Roberts operator, I defined, using NumPy, two convolution kernels that approximate the gradient magnitude of the image w.r.t x and y directions. These kernels are convolved over the images to produce the x and y gradient components, at which point the magnitude is calculated as the square root of the sum of the squares of these gradient components. This value is returned, for each function call (one for each image). For each resultant image, Otsu's method is once again employed for each image (skimage.filters package) to determine the optimal individual threshold value, and thresholding takes place for each image – this is to identify edge pixels. The result is a binary image, with black representing an edge, and white representing non-edges. See below the original images and the corresponding outputs.

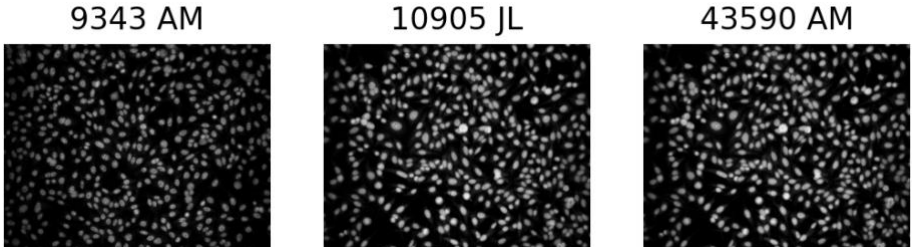


*Figure 2 - **Original** unprocessed Cell Images in Greyscale*

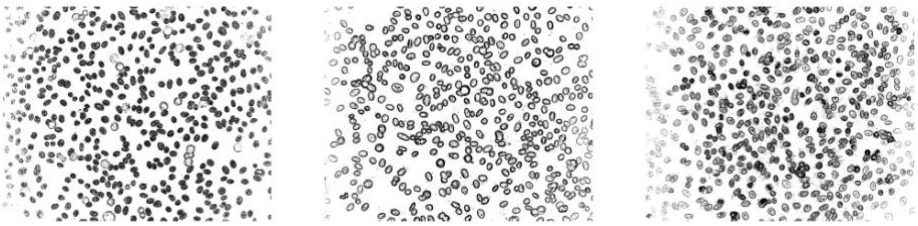


*Figure 3 - All three cell images with the **Roberts** edge detector applied*

The next test was of the Sobel edge detector. The aim of the test and the steps carried out were much the same, as Sobel and Roberts are effectively identical in operation. The difference is, the Sobel operator uses larger 3x3 convolution kernels (versus 2x2). It is considered superior for edge detection as it provides more weight to the central pixels and less weight to the surrounding pixels. In this particular test on the cell images using the Sobel operator, the difference between the output compared to the test using the Roberts operator was insignificant enough to not warrant being included on this report. To discuss the efficacy of both operators, in both cases there is false edges being detected, particularly in the case of 9343 AM wherein much of the cell 'contents' are displaying as edges. This is not so much the case with the second cells image, but is the case with 43590 AM, where it appears there are numerous false edges as well as missing edges.

The next test carried out was of the First Order Gaussian (FOG) operator. This detector applies a derivative of the Gaussian function to the images in the x and y direction using scipy.ndimage's gaussian_filter. This highlights edges by detecting where regions can be found in the image of rapid intensity variation. As with Sobel and Roberts, the function I have written takes the magnitude of these to compute the gradient magnitude of the image. Of all filters tested so far, this is the first one that has been able to reduce noise and minimise false edges, because of the effect of blurring/smoothing.

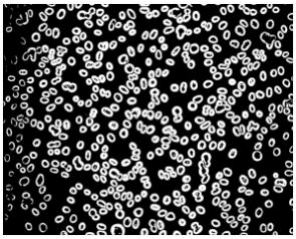


Figure 4 - FoG on 9343 AM

The Laplacian edge detector was the next to be tested. This test yielded very different results, demonstrating that the Laplacian operator is much more sensitive to noise. It detects edges regardless of orientation, and is more sensitive, but normally relies on Gaussian smoothing to enhance the base image. The steps I carried out in my script were much the same as those carried out with the Sobel and Roberts operators. The difference with the Laplacian operator is the convolution kernel used, which is a 3x3 matrix that approximates the second-order derivative of image intensity w.r.t spatial coordinates. I determined, as with the other methods, that Otsu's thresholding was the most suitable for detecting edges, specifically for finding an appropriate threshold value, yet I still ran into issues wherein much of the noise was showing as false edges. It has had the effect of emphasising some of the more subtle noise patterns in the original image, most notably a 'streakiness' caused by apparent shadows in the original 10905 JL image.

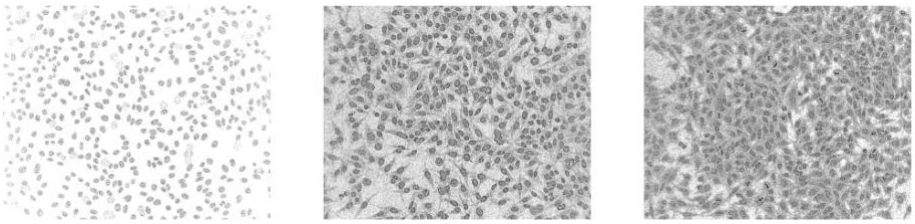


Figure 5 - Laplacian applied to all cell images

The final edge detector tested was Laplacian of Gaussian (LapGau); as used in Task 1. The steps carried out were different for this, as it involved using zero-crossing once again for thresholding (same implementation of the function as in Task 1), and Gaussian smoothing as a noise-removal technique. The standard deviation value used is 4, which was determined manually. This noise-removal technique is explained in detail as part of Task 1. The same Laplacian operator as above was used on the 'blurred' image. The smoothing also had the effect of enhancing edges. With LapGau carried out on each image, the zero_cross function was called on each output to determine which pixels in the images are edge pixels. The results are as below, demonstrating LapGau's clear superiority as an edge detector.

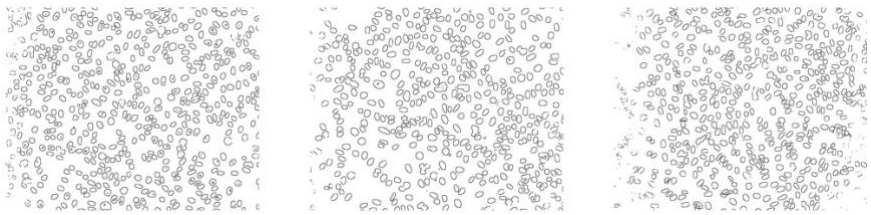


Figure 6 - Laplacian of Gaussian applied to all cell images

Task 3. For this task, I chose to implement the Canny edge detector and apply it to the three provided cell images, which is followed with a comparison with the previously-demonstrated edge detectors. It is a multi-step algorithm that is regarded for its ability to detect edges whilst minimising the effects of noise. It carries out noise reduction, gradient calculation, non-maximum suppression, and edge detection by hysteresis. The algorithm starts with Gaussian smoothing to reduce noise and minimise the effects of small intensity variations. Following this is a calculation of the gradient magnitude and direction by convolving Sobel filter kernels over the image. Non-maximum suppression refers to the stage where edges are refined to ensure that each edge pixel is a local maximum perpendicular to the direction of the gradient at that point. The fourth and final stage of the process is edge detection by hysteresis, where pixels are categorised as non-edges, strong edges, or weak "maybe" edges. Strong edges are true edges, whereas "maybe" edges are considered as such if they're connected

to existing true edges. These parts, including its ability to adapt the level of thresholding used, all work in tandem as part of a process that allows for detection of significant edges, whilst aiming to reduce errors caused by noise in the image.

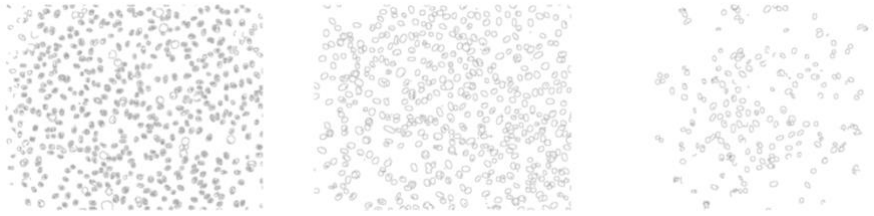


Figure 7 - Canny Edge Detector on all cell images

In my implementation, I used the skimage.feature function 'canny' into which the image and sigma (standard deviation) value is passed. As with the previous edge detector implementations, I have read in the image files in greyscale using skimage.io.imread. The results in fig. 7 show sharp, defined, and continuous/connected edges for every cell that has been detected. In the third image, it appears that a large number of the cells have been lost potentially due to an inappropriate threshold being set for this image. Where the previous detectors differ from this Canny implementation is the results are somewhat consistent, in terms of how many cells are detected and the thickness/definition of the edges. Looking at the examples shown in fig. 7, the first result indicates an output containing too many edges (all true edges in addition to many false edges), the second result arguably showing the most accurate edge detection image, with little influence from noise, and the third indicating that many cells are missing. It could be argued that Canny takes a lead ahead of the Sobel and Roberts operators with regards to performance and efficacy, given that it is a multi-stage process that refines and enhances the overall process of edge detection in comparison to the relatively basic single-stage procedures that Roberts and Sobel use. It also far exceeds the demonstrated instance of the Laplacian edge detector on the cell images, as the high-level of noise in most of the images was a point of failure that the Laplacian filter is not designed to deal with. On the other hand, First Order Gaussian demonstrated very reliable edge detection, with the results showing very few false edges caused by noise, and is overall a simpler implementation than the Canny detector. One might argue FOG falls ahead for this reason, but most of all a strong argument can be made for the Laplacian of Gaussian detector, which provides an output in which the edges are extremely well defined and there is minimal noise-induced false edges, and is very simple to implement, understand, and use. Whilst Laplacian of Gaussian is extremely effective at maximising the number of true edges it can capture, at the expense of mistaking noise for edges, Canny results in no background noise (ignoring internal structures of cells which show as edges), but appear to miss many true edges that it should have detected.

Task 4. TPR, 1-FPR (TNR), to 4 decimal places

	SOBEL	ROBERTS	FOG	LAPLACIAN	LAPGAU	CANNY
9343 AM	0.8333	0.7461	0.8281	0.8827	0.5217	0.3089
	0.8671	0.8756	0.7887	0.0467	0.9627	0.9651
10905 JL	0.8951	0.8601	0.9321	0.5869	0.6083	0.3503
	0.8871	0.8852	0.7717	0.2710	0.9676	0.9875
43590 AM	0.7310	0.6900	0.8140	0.5895	0.4802	0.0781
	0.8797	0.8659	0.7934	0.3048	0.9623	0.9944

Figure 8 - Table showing the sensitivity and specificity values for the six tested edge detectors used for the three cell images

The aim of this task was to carry out ROC analysis on the on the results to determine the sensitivity and specificity values of each edge detector. To accomplish this, I wrote a routine for each edge detector that reads in the ground truth images, passes the filtered and corresponding ground truth image into a function which calculates the number of true positives, false positives, true negatives, and false negatives. True Positive Rate (recall) and True Negative Rate are calculated from these. The results indicate that the sensitivity of FoG was the highest, at an average of 86%, meaning it is shown to be the detector with the highest proportion of correctly identified positive instances, also with an average specificity of approximately 78%. Sobel also performed well, with an average sensitivity across the three images of around 82%, but a higher specificity of 88%, compared to FoG. Predictably, Laplacian performed poorly, with an average specificity across the three images of just 21%; meaning the detector picked up a high proportion of false positives (edges detected where there are none). This would be a result of the lack of smoothing/other noise-removal techniques. On the other hand, Laplacian of Gaussian shows a surprisingly low sensitivity of approximately 54%, but one of the highest specificity values a remarkable 96%. Finally, Canny appeared to perform very poorly with regards to sensitivity, at just 26%, but with a remarkably high specificity of 98%; meaning Canny struggled to identify true edges, but did extremely well at filtering out noise, and correctly ignored the majority of non-edges. Both Laplacian of Gaussian and Canny employ Gaussian smoothing for noise-removal, clearly showcasing its efficacy through the comparatively high specificity values that were obtained. Overall, the edge detector that performed best in determining where the true edges lie was FoG, also notably with a modestly high specificity. In terms of specificity, Canny came out on top, as the edge detector that was able to filter out the most noise and minimise false positives.

Supplementary Data:

There is one Python script for Task 1, five scripts for Task 2 (for each of the five edge detectors), and one script for Task 3 (Canny). Each script also contains 'common code' that include the necessary package/library imports, the lines that read in the cell images (and ground truths), the code that uses Matplotlib to display the images, and the ROC analysis function.

Common Code:

```
import numpy as np
from matplotlib import pyplot as plt
import skimage
import scipy

image1 = skimage.io.imread('Data-2/cells/9343 AM.bmp', as_gray=True)
image2 = skimage.io.imread('Data-2/cells/10905 JL.bmp', as_gray=True)
image3 = skimage.io.imread('Data-2/cells/43590 AM.bmp', as_gray=True)

imageGT1 = skimage.io.imread('Data-2/cells/9343 AM Edges.bmp', as_gray=True)
imageGT2 = skimage.io.imread('Data-2/cells/10905 JL Edges.bmp', as_gray=True)
imageGT3 = skimage.io.imread('Data-2/cells/43590 AM Edges.bmp', as_gray=True)


def roc(filtered, groundTruth):
    # Convert ground truth image to binary
    groundTruth = np.logical_not(groundTruth/255)

    # True Positive, False Positives, True Negatives, False Negatives
    # all to be used for calculating TPR and FPR
    TP = np.sum((filtered == 1) & (groundTruth == 1))
    FP = np.sum((filtered == 1) & (groundTruth == 0))
    TN = np.sum((filtered == 0) & (groundTruth == 0))
    FN = np.sum((filtered == 0) & (groundTruth == 1))

    # Compute TPR and FPR
    tpr = TP / (TP + FN) if (TP + FN) != 0 else 0
    fpr = FP / (FP + TN) if (FP + TN) != 0 else 0

    print(f'Sobel: TPR {tpr} FPR {1-fpr}')

# ....
# this is where the implementations of the individual edge detectors would be found
# ....

plt.figure(dpi=200)

# the 'thresholded' variables are general names that refer to the variables
# containing the final edge detection image results for each implementation of an edge detector

plt.subplot(1,3,1)
plt.imshow(thresholded_1, cmap='gray')
plt.axis('off')

plt.subplot(1,3,2)
plt.imshow(thresholded_2, cmap='gray')
plt.axis('off')

plt.subplot(1,3,3)
plt.imshow(thresholded_3, cmap='gray')
plt.axis('off')
```

Task 1's Laplacian of Gaussian on Shakey Image

```
shakey_image = skimage.io.imread('Data-2/shakey.png', as_gray=True)

def zero_cross(image):
    z_c_image = np.zeros(image.shape)
    thresh = np.absolute(image).mean() * 0.75
    h,w = image.shape
    for y in range(1, h - 1):
        for x in range(1, w - 1):
            patch = image[y-1:y+2, x-1:x+2]
            p = image[y, x]
            maxP = patch.max()
            minP = patch.min()
            if (p > 0):
                zeroCross = True if minP < 0 else False
            else:
                zeroCross = True if maxP > 0 else False
            if ((maxP - minP) > thresh) and zeroCross:
                z_c_image[y, x] = 1
    return z_c_image
```

```
def find_optimal_sigma(image):
    # Define a range of sigma values to experiment with
    sigma_values = np.arange(0.5, 8.0, 0.1)  # Adjust range and step size as needed

    # Initialize lists to store threshold values and LoG-filtered images
    threshold_values = []
    filtered_images = []

    # Compute LoG-filtered images for each sigma value and apply Otsu's method
    for sigma in sigma_values:
        # Apply Gaussian blur
        blurred_image = skimage.filters.gaussian(image, sigma=sigma)
        # Apply LapGau filter
        log_filtered_image = skimage.filters.laplace(blurred_image)
        # Apply Otsu's method to determine threshold value
        threshold_value = skimage.filters.threshold_otsu(log_filtered_image)
        # Store threshold value and filtered image
        threshold_values.append(threshold_value)
        filtered_images.append(log_filtered_image)

    # Determine the optimal sigma value based on the maximum threshold value
    optimal_sigma_index = np.argmax(threshold_values)
    optimal_sigma = sigma_values[optimal_sigma_index]
    optimal_filtered_image = filtered_images[optimal_sigma_index]

    return optimal_sigma, optimal_filtered_image

optimal_sigma, optimal_sigma_image = find_optimal_sigma(shakey_image)
z_c_filtered = zero_cross(optimal_sigma_image)
```

Task 2's Sobel Edge Detector on Cell Images

```
def sobel_edge_detector(image, threshold=0.1):
    # Define Sobel operators
    sobel_x = np.array([[1, 0, -1],
                        [2, 0, -2],
                        [1, 0, -1]])
    sobel_y = np.array([[1, 2, 1],
                        [0, 0, 0],
                        [-1, -2, -1]])

    # # Apply Sobel operators to compute gradients
    gradient_x = scipy.signal.convolve2d(image, sobel_x, mode='same')
    gradient_y = scipy.signal.convolve2d(image, sobel_y, mode='same')

    # # Compute magnitude of gradients
    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

    return gradient_magnitude

sobel_edges_image1 = sobel_edge_detector(image1, 0.15)
sobel_edges_image2 = sobel_edge_detector(image2, 0.15)
sobel_edges_image3 = sobel_edge_detector(image3, 0.15)

th_1 = skimage.filters.threshold_otsu(sobel_edges_image1)
th_2 = skimage.filters.threshold_otsu(sobel_edges_image2)
th_3 = skimage.filters.threshold_otsu(sobel_edges_image3)

thresholded_1 = sobel_edges_image1 > th_1
thresholded_2 = sobel_edges_image2 > th_2
thresholded_3 = sobel_edges_image3 > th_3
```

Task 2's Roberts Edge Detector on Cell Images

```
def roberts_edge_detector(image):
    # Define convolution matrices
    roberts_x = np.array([[1,0],
                          [0,-1]])
    roberts_y = np.array([[0,1],
                          [-1,0]])

    #Apply Roberts operators to compute gradients
    edges_x = scipy.ndimage.convolve(image, roberts_x)
    edges_y = scipy.ndimage.convolve(image, roberts_y)

    # Take magnitude of horiz and vert components
    magnitude = np.sqrt(edges_x**2 + edges_y**2)

    return magnitude

roberts_edges_image1 = roberts_edge_detector(image1)
roberts_edges_image2 = roberts_edge_detector(image2)
roberts_edges_image3 = roberts_edge_detector(image3)

th_1 = skimage.filters.threshold_otsu(roberts_edges_image1)
th_2 = skimage.filters.threshold_otsu(roberts_edges_image2)
th_3 = skimage.filters.threshold_otsu(roberts_edges_image3)

thresholded_1 = roberts_edges_image1 > th_1
thresholded_2 = roberts_edges_image2 > th_2
thresholded_3 = roberts_edges_image3 > th_3
```

Task 2's First Order Gaussian on Cell Images

```
def first_order_gaussian(image, sigma=3):
    # Apply Gaussian filter in x and y direction
    image_gaussian_x = scipy.ndimage.gaussian_filter(image, sigma, order=(0, 1))
    image_gaussian_y = scipy.ndimage.gaussian_filter(image, sigma, order=(1, 0))

    # Take magnitude of gradient using x and y directional gradient components
    magnitude_gaussian = np.sqrt(np.square(image_gaussian_x) + np.square(image_gaussian_y))

    return magnitude_gaussian

# Run FoG on the images
fog_image_1 = first_order_gaussian(image1)
fog_image_2 = first_order_gaussian(image2)
fog_image_3 = first_order_gaussian(image3)

#  Use Otsu's method to determine optimal threshold level
th_1 = skimage.filters.threshold_otsu(fog_image_1)
th_2 = skimage.filters.threshold_otsu(fog_image_2)
th_3 = skimage.filters.threshold_otsu(fog_image_3)

# Threshold the filtered images to create binary images
thresholded_1 = fog_image_1 > th_1
thresholded_2 = fog_image_2 > th_2
thresholded_3 = fog_image_3 > th_3
```

Task 2's Laplacian on Cell Images

```
def laplacian_edge_detector(image):
    # convolution matrix representing approximation of 2nd order derivative
    laplacian_kernel = np.array([[0,1,0],
                                 [1,-4,1],
                                 [0,1,0]])

    laplacian_image = scipy.ndimage.convolve(image, laplacian_kernel)

    return laplacian_image

laplacian_edge_image1 = laplacian_edge_detector(image1)
laplacian_edge_image2 = laplacian_edge_detector(image2)
laplacian_edge_image3 = laplacian_edge_detector(image3)

th_1 = skimage.filters.threshold_otsu(laplacian_edge_image1)
th_2 = skimage.filters.threshold_otsu(laplacian_edge_image2)
th_3 = skimage.filters.threshold_otsu(laplacian_edge_image3)

thresholded_1 = laplacian_edge_image1 > th_1
thresholded_2 = laplacian_edge_image2 > th_2
thresholded_3 = laplacian_edge_image3 > th_3
```

Task 2's Laplacian of Gaussian on Cell Images

```
def laplacian_of_gaussian(image, sigma=4):
    # Apply Gaussian blur to the image
    blurred_image = scipy.ndimage.gaussian_filter(image, sigma)

    laplacian_kernel = np.array([[0,1,0],
                                 [1,-4,1],
                                 [0,1,0]])
    laplacian_image = scipy.ndimage.convolve(blurred_image, laplacian_kernel)
    # laplacian_image = skimage.filters.laplace(blurred_image)

    return laplacian_image

op_image_1= laplacian_of_gaussian(image1)
op_image_2 = laplacian_of_gaussian(image2)
op_image_3 = laplacian_of_gaussian(image3)

z_c_image_1 = zero_cross(op_image_1)
z_c_image_2 = zero_cross(op_image_2)
z_c_image_3 = zero_cross(op_image_3)
```

Task 3's Canny edge detector on Cell Images

```
def canny_edge_detector(image, sigma=1):
    # Apply Canny edge detection
    edges = skimage.feature.canny(image, sigma=sigma)

    return edges

edges_image1 = canny_edge_detector(image1)
edges_image2 = canny_edge_detector(image2)
edges_image3 = canny_edge_detector(image3)

th_1 = skimage.filters.threshold_otsu(edges_image1)
th_2 = skimage.filters.threshold_otsu(edges_image2)
th_3 = skimage.filters.threshold_otsu(edges_image3)

thresholded_1 = edges_image1 > th_1
thresholded_2 = edges_image2 > th_2
thresholded_3 = edges_image3 > th_3
```